Homework 1 Selected Solutions
APPM/MATH 4650 Fall '20 Numerical Analysis

Due date: Friday, September 4, before 5 PM, via Gradescope. **Instructor**: Prof. Becker
Theme: Introduction to floating point computations; stability; conditioning

solutions version 9/9/2020

Problem 1: Consider the polynomial

$$p(x) = (x-2)^9 = x^9 - 18x^8 + 144x^7 - 672x^6 + 2016x^5 - 4032x^4 + 5376x^3 - 4608x^2 + 2304x - 512.$$

Note: you can get these coefficients using Matlab's `poly` or Python's `numpy.poly` and specifying that 2 is a root with multiplicity 9; this saves you having to type them in.

a) Produce a plot that shows the evaluation of p for 100 equispaced points in the range $[1.92, 2.08]$, using the following 4 algorithms for evaluating a polynomial (overlay all 4 results on the same plot):

   i. Make your own algorithm to naively evaluate the polynomial using its coefficients

   ii. Use the compact form $p(x) = (x-2)^9$

   iii. Implement Horner's rule

   iv. Use a software library (e.g., Matlab's `polyval` or Python's `numpy.polyval`).

b) Comment on the similarities and differences, and discuss which algorithm you think is most correct, and what possible sources of numerical error might be.

Solution:

a) Here's a solution in matlab:

```
trueRoot    = 2;
degree      = 9;
p           = poly( repmat(trueRoot,1,degree) );
f    = @(x) (x-trueRoot).^degree;     % compact
g    = @(x) polyval( p, x );          % Matlab's builtin
h    = @(x) (p*( x(:).^(9:-1:0) )' ); % naive
j    = @(x) HornersRule( p, x );      % own implementation of Horner's
% x   = linspace( -500, 500, 1e2 );  % boring (but sanity check)
x    = linspace( 1.92, 2.08, 1e2 );

plot( x, f(x), 'linewidth',2','DisplayName','compact' );
hold all
plot( x, g(x), 'linewidth',2','DisplayName','Matlab''s polyval' );
plot( x, h(x), '--','linewidth',2','DisplayName','naive' );
plot( x, j(x), '--','linewidth',2','DisplayName','Own implementation of
    Horner' );
ylim([-.3,.3]*1e-10); xlim([1.92,2.08]);
hl=legend('location','northwest'); hl.FontSize = 22; set(gca,'FontSize',20);
\begin{lstlisting}

and the implementation of \texttt{HornersRule} is
\begin{lstlisting}  % [style=Python]
```

©2020 Stephen Becker

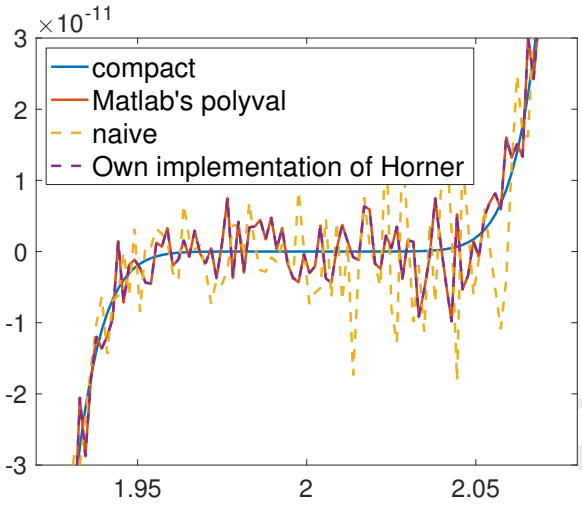

Figure 1: The four methods of Problem 1(a); the "compact" representation is by far the most accurate

```
function y = HornersRule( p, x )
    y   = p(1);
    for i = 2:length(p)
    y   = (y + p(i)).*x; % do .* so that it can vectorize
    end
```

or

```
function y = HornersRule( p, x )
    y   = p(1)*x;
    for i = 2:(length(p)-1)
        y   = (y + p(i)).*x; % do .* so that it can vectorize
    end
    y   = y + p(end);
```

Here's a solution in Python:

```
import numpy as np

# We're not going to use sympy for math, but it has a nice
# feature that makes pretty output:
from sympy import init_printing
init_printing()

trueRoot = 2
degree   = 9
p        = np.poly( degree*(trueRoot,) )#syntax: 3*(a,b) gives (a,b,a,b,a,b)
x        = np.linspace( 1.92, 2.08, int(2e1) )  # np.arange() works too

f  = lambda x : (x-trueRoot)**degree
g  = lambda x : np.polyval( p, x )
def naivePolynomialEvaluation( p, x ):
xx  = x.copy() # things like np.full_like  are also useful
#xx  = x  # No!! Major bug
y   = p[-2]*xx + p[-1]
for coeff in reversed(p[:-2]):
```

©2020 Stephen Becker

```
xx *= x  # if you didn't make a copy, this has side-effects!
y  += coeff*xx
return y
def HornersRule( p, x ):
y = p[0]
for coeff in p[1:]:
y = y*x + coeff
return y

# Plotting
import matplotlib.pyplot as plt
import matplotlib as mpl
%matplotlib inline
plt.style.use('seaborn-ticks')
mpl.rcParams.update({'font.size': 18})
mpl.rcParams['mathtext.fontset'] = 'cm'
mpl.rcParams['lines.linewidth'] = 2.0

plt.figure();
fig, ax = plt.subplots();
fig.set_size_inches(10,6)
ax.plot(x, f(x),label='compact')
ax.plot(x, g(x),label=Numpy's polyval,linewidth=1)
ax.plot(x, naivePolynomialEvaluation(p,x),'--',label=naive)
ax.plot(x, HornersRule(p,x),'--',label=Horner's,linewidth=3)
plt.xlabel(r'$x$',fontsize=24)
plt.ylabel(r'$y$',fontsize=24)
ax.legend(fontsize=14,frameon=True,loc=9);

# The bbox_inches fixes some cropping issues that chopped of the labels
plt.savefig('HW01_plot1.pdf',bbox_inches='tight')
```

b) Discussion: the compact representation is most accurate by far. The naive implementation has a lot of addition and subtraction of possibly large terms, and these can ruin the accuracy. Horner's rule is slightly better (barely). The Matlab/Numpy `polyval` functions appear to give exactly the same answer as Horner's rule, so this suggests that they use Horner's rule.

Problem 2: How would you perform the following calculations to avoid cancellation? Justify your answers.

a) Evaluate $f(x) = \sqrt{x+1} - 1$ for $x \approx 0$.

Solution:

This has bad subtractive cancellation. The trick is to rationalize the numerator, as we did in `Ch1_QuadraticFormula.ipynb`. Write

$$\sqrt{x+1} - 1 = \frac{\sqrt{x+1}+1}{\sqrt{x+1}+1} \cdot \left(\sqrt{x+1} - 1\right) = \frac{(x+1) - 1}{\sqrt{x+1}+1} = \boxed{\frac{x}{\sqrt{x+1}+1}}$$

which is now stable and accurate to machine precision for $x \approx 0$. The trick is that we did part of the subtractive cancellation, $1 - 1$, analytically, rather than do that part on the computer.

b) ~~Evaluate $f(x) = \sin(2x) - \sin(2a)$ for $x \approx a$. *Hint*: try a trig identity.~~
Evaluate $f(x) = \sin(2(x+a)) - \sin(2a)$ for $x \approx 0$. *Hint*: try a trig identity.

©2020 Stephen Becker

Solution:

There can be devastating cancellation via direct computation, so we need to fix. We want a trig identity that collects $2(x+a)-2a$ *inside* a trig function, since then we can cancel the $2a-2a$ by hand.

We can use this trig identity[1]

$$\sin\theta \pm \sin\varphi = 2\sin\left(\frac{\theta \pm \varphi}{2}\right)\cos\left(\frac{\theta \mp \varphi}{2}\right)$$

with $\theta = 2(x+a)$ and $\varphi = 2a$ giving

$$f(x) = 2\sin\left(\frac{2(x+a)-2a}{2}\right)\cos\left(\frac{2(x+a)+2a}{2}\right) = \boxed{2\sin(x)\cos(x+2a)}$$

and $\sin(x)$ and $\cos(x+2a)$ are stable for $x \approx 0$.

Don't do this trig identity $\sin(\alpha \pm \beta) = \sin\alpha\cos\beta \pm \cos\alpha\sin\beta$ unless you do further simplification, since this identity doesn't remove the subtractive cancellation.

Problem 3: a) Suppose a sequence has the form

$$x_n = Cn^{-\alpha} \tag{1}$$

for some constants C and α. If you plot n on the x-axis and x_n on the y-axis, this does not form a straight line. On what kind of plot would this form a straight line? (e.g., logarithmic scaling on the x-axis? on the y-axis? on both axes?). Justify your answer.

Solution:

Taking the log of both sides of the equation, we have $\log(x_n) = \log(C) - \alpha\log(n)$, so if we make a change of variables with $y = \log(x_n)$ and $x = \log(n)$, then this is a straight line in an $x-y$ graph.. So this is a straight line on a $\boxed{\text{log-log plot}}$ of x_n (on the y-axis) and n (on the x-axis). Note that it doesn't matter what base of the logarithm we take (that would affect the slope and intercept values, but not the fact that it's a straight line).

b) Repeat the question above, but for a sequence of the form

$$x_n = D\rho^n \tag{2}$$

for some constants D and $\rho < 1$.

Solution:

Taking the log of both sides of the equation, we have $\log(x_n) = \log(D) + n\log(\rho)$, so letting $y = \log(x_n)$ and $x = n$, this is a straight line in an $x-y$ graph. So this is a straight line if the $\boxed{\text{y-axis is logarithmic and the x-axis is linear}}$.

c) Suppose you are given the first 10 terms of a sequence (x_n):

```
5.6000  4.4800  3.5840  2.8672  2.2938  1.8350  1.4680  1.1744  0.9395  0.7516
```

Estimate (you cannot *prove* anything, since you've only seen the first 10 terms) whether this sequence converges sublinearly, linearly, superlinearly, or quadratically; and explain why you think so. If the sequence fits the form of Eq. (1) or Eq. (2), find the values of the parameters (either (C,α) or (D,ρ)).

[1] See the first "Sum-to-product" identity at wikipedia: trig identities#Product-to-sum_and_sum-to-product_identities

©2020 Stephen Becker

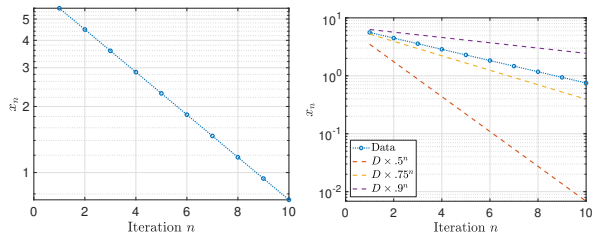

Figure 2: Plots for problem 3c (linear convergence)

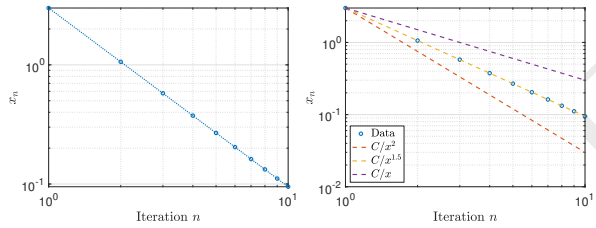

Figure 3: Plots for problem 3d (sublinear convergence)

Solution:

Based on being a straight line on a $\log(y) \times x$ plot (see Fig. 2), we suspect this is $\boxed{\text{linear convergence}}$. Then trying to fit to the form of Eq. (2), we can either do a least-squares regression (e.g., fit a stright line to $\log(x_n)$ and n, using `polyfit` for example), or since this is an academic example and fits the equation perfectly, we can just pick any two points (n, x_n) and (m, x_m) and then algebraically solve Eq. (2). The answer is $\boxed{D = 7 \text{ and } \rho = 0.8}$.

Specifically, solve $x_n = D\rho^n$ and $x_m = D\rho^m$ for D and ρ by taking $x_n/x_m = \rho^{n-m}$ to solve for ρ, and then $D = x_n/\rho^n$.

d) Repeat the question above, but for the following sequence

```
3.0000   1.0607   0.5774   0.3750   0.2683   0.2041   0.1620   0.1326   0.1111   0.0949
```

Solution:

Since this is linear on a log-log plot (see Fig. 3), we suspect it is $\boxed{\text{sub-linear}}$ convergence of the form Eq. (1). As above, we then pick two points n, m to solve for the constants, finding $\boxed{C = 3,\ \alpha = 1.5}$.

Problem 4: At what rate (i.e., use big-O notation) does $\frac{1}{1-h} - h - 1$ converge to 0 as $h \to 0$?

Solution:

The Taylor series of $\frac{1}{1-h}$ at $h = 0$ is

$$\frac{1}{1-h} = 1 + h + h^2 + h^3 + \dots$$

so

$$\frac{1}{1-h} - h - 1 = h^2 + h^3 + \dots = \boxed{\mathcal{O}(h^2)}.$$

Note that for these problems, we almost always look for convergence rates that are $\mathcal{O}(h^p)$ for some integer p (it's not very common to look for rates that are, say, $\mathcal{O}(\sin(h))$ or $\mathcal{O}(1/(1-h))$, for example... in this case, both these expressions can be further simplified using their Taylor series).

©2021 Stephen Becker

Problem 5: Let $f(x) = e^x - 1$

a) What is the relative condition number $\kappa_f(x)$? Are there any values of x for which this is ill-conditioned?

Solution:

Compute

$$\kappa_f(x) = \left| \frac{x}{f(x)} f'(x) \right| = \left| \frac{xe^x}{e^x - 1} \right|$$

As $x \to \infty$, then $\frac{e^x}{e^x - 1} \to 1$, so $\kappa_f(x) \approx x$ for large x. This is moderately ill-conditioned.

As $x \to 0$, we can compute $\kappa_f(x)$ using L'Hôpital's rule, or for a quick and dirty way,

$$\lim_{x \to 0} \left| \frac{xe^x}{e^x - 1} \right| = \lim_{x \to 0} \left| \frac{x}{e^x - 1} \right| \cdot \lim_{x \to 0} |e^x| = \lim_{x \to 0} \left| \frac{x}{x + \mathcal{O}(x^2)} \right| \cdot 1 = 1$$

where we used the Taylor series of e^x in the denominator.

So in particular, $\boxed{\text{for } x \approx 0, \text{ this function is } \textit{not} \text{ ill-conditioned}}$.

b) Consider computing $f(x)$ via the following algorithm:

1: $y \leftarrow e^x$
2: **return** $y - 1$

(That is, we are computing f just the way it's written, doing $e^x - 1$). Is this algorithm stable? Justify your answer

Solution:

No, this algorithm is $\boxed{\text{not stable}}$, due to the 2nd step. Let's work out the condition number of each step. We're essentially computing f via $f(x) = g(h(x))$ where $h(x) = e^x$ and $g(y) = y - 1$. Then

$$\kappa_h(x) = \left| \frac{x}{e^x} e^x \right| = |x|$$

which is no worse than κ_f for large x, and is not an issue at all for $x \approx 0$.

Also,

$$\kappa_g(y) = \left| \frac{y}{y - 1} 1 \right|$$

which blows up as $y \to 1$ which is a problem because as $x \to 0$, then $y = e^x \to 1$.

c) Let x have the value $9.999999995000000 \times 10^{-10}$, in which case $f(x)$ is equal to 10^{-9} up to 16 decimal places. How many correct digits does the algorithm listed above give you? Is this expected?

Solution:

This is a straightforward implementation on the computer. We find about 7 accurate digits. This makes sense because we're not stable and expect to lose about 9 digits.

How do we know we expect to lose about 9 digits here? We have

$$\text{relative error} \approx \epsilon_{\text{machine}} \cdot \kappa$$

for each step (this is approximate since it's only exactly true as $\epsilon_{\text{machine}} \to 0$, so applying this to the last step, with $\epsilon_{\text{machine}} = 2.2 \cdot 10^{-16}$ and $\kappa = \kappa_g(e^x)$ for $x \approx 10^{-9}$ so (via Taylor series) $e^x \approx 1 + x \approx 1 + 10^{-9}$ and $\kappa \approx (1 + 10^{-9})/10^{-9} \approx 10^9$, so

$$\text{relative error} \approx \epsilon_{\text{machine}} \cdot \kappa \approx 10^{-16} \cdot 10^9 = 10^{-7}.$$

When we ask for the number of digits we lose, this is $-\log_{10}(\kappa)$.

©2020 Stephen Becker

Note that because we are using the specified algorithm, we use the largest relative condition number that we find in the algorithm steps, so κ_g. We are *not* asking for κ_f here. This can be confusing, in particular because even though $\kappa_f(0) = 1$ (so not ill-conditioned), when you *numerically evaluate* $\kappa_f(0)$ you'll get roundoff errors that might tell you $\kappa_f(0) \approx 10^{-8}$, which is not true. The naive algorithm to numerically estimating the condition number is itself unstable!

d) Find a polynomial approximation of $f(x)$ that is accurate to 16 digits for all $|x| \leq 10^{-9}$ and prove your answer. *Hint*: use Taylor series, and remember that 16 digits of accuracy is a *relative* error, not an *absolute* one.

Solution:

We want a relative accuracy of 10^{-16}, meaning an approximation $\hat{f}(x)$ with

$$\frac{|\hat{f}(x) - f(x)|}{|f(x)|} \leq 10^{-16}$$

For $|x| \leq 10^{-9}$, using Taylor's theorem we can write $f(x) = f(0) + f'(0)x + f''(\xi)x^2/2$ for $\xi \in [0, x)$ (hence $|\xi| \leq 10^{-9}$). Plugging in these values, for $|x| \leq 10^{-9}$ and $|\xi| \leq 10^{-9}$,

$$\frac{|\hat{f}(x) - f(x)|}{|f(x)|} = \frac{|(e^0 - 1) + e^0 x + e^\xi x^2/2|}{|e^x - 1|} \leq \frac{|x + 2(10^{-9})^2/2|}{|e^x - 1|} \leq \frac{|x + 10^{-18}|}{|e^x - 1|}$$

where we used $e^\xi \leq 2$ for $|\xi| \leq 10^{-9}$. We need an upper bound on the denominator, and you can do this but it's annoying, and we'll just note that $e^x - 1 \approx x$ for very small x, and thus we can bound the denominator (loosely) by $|x|$.

Thus if we choose $\hat{f}(x) = x$, the relative error is bounded by $\frac{10^{-18}}{10^{-9}} = 10^{-9}$, which is not enough accuracy.

So, we can do the same thing for the second order Taylor series $\boxed{\hat{f}(x) = x + x^2/2}$ which has error term $|e^\xi x^3/3!| \leq 10^{-27}/3$, so now, even when we normalize by $f(x) \approx 10^{-9}$, we still have about 10^{-18} relative accuracy, which gives us more than 16 correct digits.

e) Evaluate your polynomial approximation at the same value of x as before. How many digits of precision do you have?

Solution:

All digits (as many as we had in the "true" answer), i.e., at least 16. If you appear to have an infinite number of correct digits, that's misleading, because the "true" solution I gave you is not infinitely accurate.

Make sure that when you evaluate accuracy, you use the reference solution I gave you. If you compute $e^x - 1$ as the "true solution", that is not right! Even though that formula looks correct, we just showed that it is unstable, so it has high relative error. So it is *not* your reference solution. As we see, our Taylor series is *more accurate* than what seems like the "correct" way to program it.

Also, we should note that we've been discussing two types of errors:

i. Numerical roundoff error. The algorithm $e^x - 1$ is unstable and has high roundoff error. There is no "mathematical" error. This error has everything to do with computers and the effects of finite accuracy.

ii. Approximation error (think of this as error in the "math"), which would be true even if we had infinite accuracy representations on the computer. The Taylor series error from part (d) is this kind of error. This error has nothing to do with computers.

Of course, for scientific computations, we care about *both* types of errors! In part (d), we are allowing for a small amount of approximation error in order to have smaller roundoff error. We'll come back to this when we discuss estimating derivatives.

f) [Optional] How many digits of accuracy do you have if you do a simpler Taylor series?

©2020 Stephen Becker

Solution:

For just using $e^x - 1 \approx x$, we have about 8 digits near $x = 0$. The absolute accuracy is good, but relative accuracy not so much.

g) [Fact; no work required] Matlab provides `expm1` and Python provides `numpy.expm1` which are special-purpose algorithms to compute $e^x - 1$ for $x \approx 0$. You could compare your Taylor series approximation with `expm1`.

Solution:

The output of `expm1` is identical to the first two terms in the Taylor series (i..e, the approximation we derived in part (d)), so there's a good chance that's how the `expm1` function is implemented.

©2020 Stephen Becker